A User Manual

In the following, the features of the *Symbolic Execution Debugger* and their usage is explained. Debugging can be started by selecting **Start Symbolic Execution Debugging** in the pop up menu of the target method in either the **Outline View** or the **Package Explorer**. The *Symbolic Execution Debugger* is then initialized and the target method is invoked with symbolic input values, but symbolic execution suspends before the first statement is about to be executed in order to transfer the control over execution to the user.

The *Symbolic Execution Debugger* mainly contributes functionality via four views that are explained in the following.

A.1 Symbolic Execution Debugger View

The *Symbolic Execution Debugger View* is the central component of the *Symbolic Execution Debugger*. The local tool bar offers buttons for the commands *Run*, *Step Into* and *Step Over* which allow to control symbolic execution:

Run continues symbolic execution of all explored symbolic states.

Step Into expands the current execution tree at each leave node. For a given leave node's symbolic state, *Step Into* discovers all symbolic states that are reachable from this symbolic state by continuing symbolic execution until the next Java statement is about to be executed. This simply means that the corresponding statement level execution tree is expanded one level more.

Step Over has the same effect as *Step Into* except that it does not go into method invocations.

The path condition of a selected node in the *Execution Tree View* is displayed in the middle of the *Symbolic Execution Debugger View*. The local pull down menu offers the possibility to change internal settings of the *Symbolic Execution Debugger*.

Use All Invariants allows to declare whether all class invariants which are specified as JML-Expressions in the source code should be added automatically to the precondition.

Quantifier Instantiation with Splitting specifies whether quantifier instantiations that cause the underlying proof to split up are allowed.

Show Implicit Attributes determines whether implicit attributes are displayed in the current debugging session. This affects the execution tree including path and branch conditions, but also the entire symbolic state visualization.

KeY-Prover Window shows the underlying KeY-Prover which actually performs symbolic execution. Please note, that it is not necessary to interact with the KeY-Prover.

A.2 Statement Breakpoint View

The *Symbolic Execution Debugger* makes use of so-called statement breakpoints. In contrast to line breakpoints that are provided by Eclipse, statement breakpoints are attached to entire Java statements.

Symbolic execution of a symbolic state pauses as soon as its program counter points to a statement which is equipped with a statement breakpoint. Of course, symbolic execution continues with states that have not reached a statement breakpoint. The effect of symbolically executing a Java program annotated with statement breakpoints is that some leaves of the resulting execution tree may be labeled with symbolic states which are in break mode and thus not further expanded.

The *Statement Breakpoint View* allows to attach and remove statement breakpoints and shows the current existing statements breakpoints. The *Statement Breakpoint View* consists of a list which manages the currently set breakpoints. It is possible to add a statement breakpoint by setting the cursor on the particular statement in the Java Editor and then pushing the button **Add** in the *Statement Breakpoint View*. Statements which are equipped with a breakpoint are highlighted in the Java Editor similar to the built-in line breakpoints of Eclipse. The actual appearance (for instance the color) of statement breakpoints can be configured on the **Preference** page of Eclipse. A selected statement breakpoint can be removed with the help of the button **Remove**.

A.3 Execution Tree View

Whenever symbolic execution is suspended, the explored (partial) execution tree is visualized in the *Execution Tree View*. Each node in an execution tree is associated with a symbolic state and each node is labeled with the program counter of the corresponding symbolic state. There are five types of nodes in an execution tree:

Statement Nodes represent symbolic states whose program counter points to an entire Java statement. These nodes are visualized by a blue rectangle labeled with the particular statement[1]. A double click on a node in the execution tree highlights the associated Java statement in the Java editor in order to show the current program counter in the context of the entire program.

Expression Nodes represent symbolic states whose program counter points to a Java expression. Currently there are only expression nodes for guard expressions of loops, since they are executed iteratively. An expression node is drawn as a rounded, light magenta colored rectangle labeled with a particular expression. A double click on an expression node in the execution tree highlights the associated Java expression in the source code.

Method Invocation Nodes represent symbolic states whose program counter points to a method invocation. Such nodes are visualized by a white rectangle labeled with the name of the invoked method and the particular symbolic values of the input parameters. The class from which the implementation of this method is taken and further details about the method invocation are shown as tool tip.

Method Return Nodes represent the return from method calls and they are visualized by a white rectangle labeled with `return returnValue`, whereby `returnValue` denotes the symbolic return value of the method call. Selecting a method return node highlights the method invocation node which belongs to this method return node. So the method stack of a symbolic state is implicitly represented on an execution path.

Termination Nodes indicate the termination of symbolic execution and are visualized by colored circles. A green circle indicates normal termination. On the other hand, a red circle represents abrupt termination. Selecting such a red circle highlights the exact expression in the Java Editor which caused the thrown exception. Additionally, the type of the exception is shown as a tool tip.

Branch conditions show case distinctions which are made in order to cause execution to take a particular branch. A branch condition is displayed as a tool tip of the label **BC** that is attached to particular edges. Moreover, the branch condition is also displayed in a separate compartment as a list when a particular execution tree node is selected. The path condition of a symbolic state is shown in the *Symbolic Execution Debugger View* in case that the corresponding execution tree node is selected. The local tool bar of the *Execution Tree View* consists of two buttons:

[1] Actually only the first line of the statement is shown, but the entire statement is depicted as a tool tip.

The button **Create Test Cases** generates test cases for the currently explored execution paths. The button **Run Decision Procedures** runs an external decision procedure in order find more infeasible execution paths. The decision procedure specified in the settings of the KeY-Prover is used for this purpose.

Execution tree nodes are equipped with a context menu which can be opened with a right mouse click. Such a menu offers the following items:

Run, Step Into and Step Over cause the respective command to be applied on the selected subtree.

Create Test Case Creates test cases for the execution path up to the selected node.

Visualize Draws *symbolic object diagrams* for the selected node in the *Symbolic State View*.

A.4 Symbolic State View

The *Symbolic State View* is responsible for visualizing symbolic states. A symbolic state can be visualized by selecting **Visualize** in the context menu of a node or alternatively by selecting **Visualize** in the context menu of the path condition in the *Symbolic Execution Debugger View*. A symbolic state is visualized as a set of *symbolic object diagrams*. A *symbolic object diagram* can be chosen with the sliders **Instance Configuration** and **Index Configuration**. The slider **Instance Configuration** iterates over the possible existing instances and their associations. On the other hand, the slider **Index Configuration** iterates over the possible array indices. *Symbolic Object diagrams* are generated for the prestate and poststate.

A *symbolic object diagram* for the prestate represents a set of concrete heaps that cause symbolic execution to follow the particular associated execution path through the control flow graph. Each *symbolic object diagram* for a prestate is associated with a *symbolic object diagram* for the poststate which shows the actual heap that results from executing the associated execution path. It is possible to switch between *symbolic object diagrams* for the prestate and poststate with the buttons `Poststate` and `Prestate`.

The entities of a *symbolic object diagram* are layouted automatically, nevertheless it is possible to move them manually via drag and drop.

A.5 Test Case Generation

Automatically generated test cases are stored in a special Java project `testFiles` which is created if it does not exist yet. After a test file is created, the name of

the new test file is shown in a message dialog. In Eclipse, a JUnit test case can be executed by clicking **Run** → **Run as** → **JUnit Test** in the context menu of the test file in the *Package Explorer*. The result of a test run including an error trace are shown in the JUnit view. The *Symbolic Execution Debugger* hooks into the execution of test cases and as soon as a test case fails, the corresponding execution path in the execution tree is highlighted. Note, that the mapping from test runs to execution paths is not persistent and gets lost as soon as the *Symbolic Execution Debugger* is initialized again.

tree and visualizes it in the *Execution Tree View* (see Chapter 5). It is then possible to visualize the symbolic heap for a particular execution tree node in the Symbolic State View (see Chapter 6).

3.5 A First Example: `PayCard`

The purpose of this example is to give a first impression of the above explained views and the functionality of the *Symbolic Execution Debugger*. A detailed description of the provided features is given in the Appendix A.

The features of the *Symbolic Execution Debugger* are demonstrated with the `PayCard` example [ER05] which is shipped with the KeY-Prover. The class `LogFile` in the `PayCard` example keeps track of a number of processed transactions by storing the balances in the end of the transactions. The method `getMaximumRecord()` in this class returns the tracked log entry (instances of `LogRecord`) with the greatest balance. The source code of method `getMaximumRecord()` is listed in Figure 3.3.

```
/*@ public normal_behavior
  @    ensures (\forall int i; 0 <= i && i<logArray.length;
  @             logArray[i].balance <= \result.balance);
  @    diverges true;
  @ */
public /*@pure@*/ LogRecord getMaximumRecord(){
    LogRecord max = logArray[0];
    int i=1;
    while(i<logArray.length){
        LogRecord lr = logArray[i++];
        if (lr.getBalance() > max.getBalance()){
            max = lr;
        }
    }
    return max;
}
```

Figure 3.3: Source code of method `getMaximumRecord()`.

Figure 3.4 shows a screenshot of the Eclipse workbench. On the bottom of this workbench, the *Statement Breakpoint View* of the *Symbolic Execution Debugger* is currently open. Furthermore, the class `LogFile` of the `PayCard` example which can be found in [ER05] is loaded into a Java Editor. Currently, an active statement

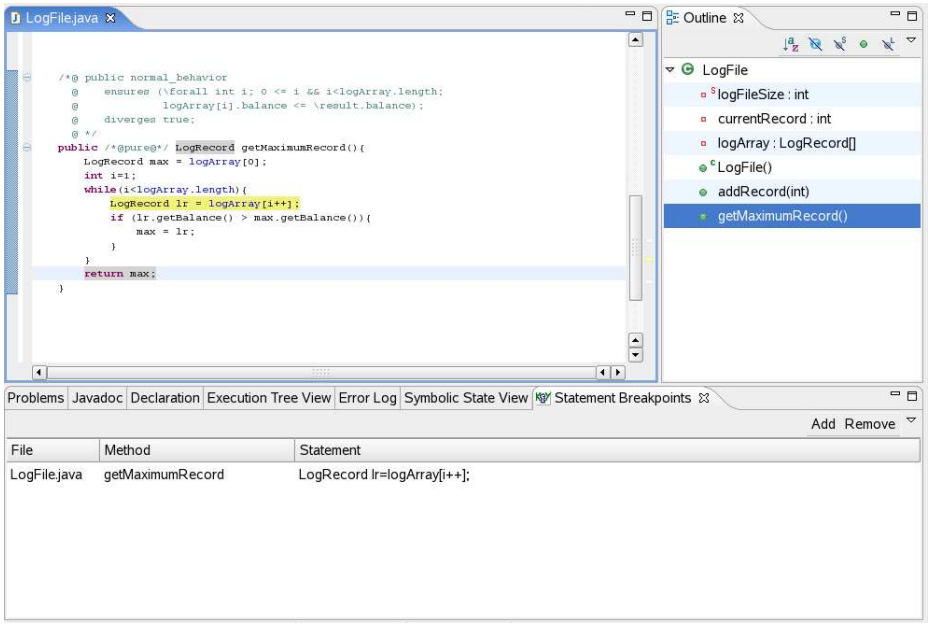

Figure 3.4: *Statement Breakpoint View* of the *Symbolic Execution Debugger*.

breakpoint is set on the Java Statement `LogRecord lr = logArray[i++];`. Statement breakpoints are highlighted in the Java editor and displayed in a list in the *Statement Breakpoint View*.

Debugging of `getMaximumRecord()` can be started by selecting **Start Symbolic Execution Debugging** in the method's context menu in either the **Outline View** or the **Package Explorer**. The execution tree resulting from symbolic execution of `getMaximumRecord()` is shown in Figure 3.5. The left-most execution tree node is surrounded by a red rectangle which indicates that the symbolic state represented by this node has reached a statement breakpoint (in this case `LogRecord lr = logArray[i++];`). The execution tree is not expanded at this node anymore, unless the statement breakpoint is removed.

White nodes represent method invocations or returns from a method call. Rounded, light magenta colored rectangles represent the evaluation of expressions, in this example the only expression is the loop condition of the `while`-loop. A blue node represents the execution of a Java statement, i.e. such a node shows the program counter of the associated symbolic state. Finally, the branch condition compartment currently shows the branch condition

$\texttt{selfLogFile.logArray} \neq \texttt{null}$ *and* $\texttt{selfLogFile.logArray.length} \leq 1$

for the selected node `return max;` in the execution tree. Branch conditions are also shown as tool tips when hovering over the label **BC**. The right-most leave node

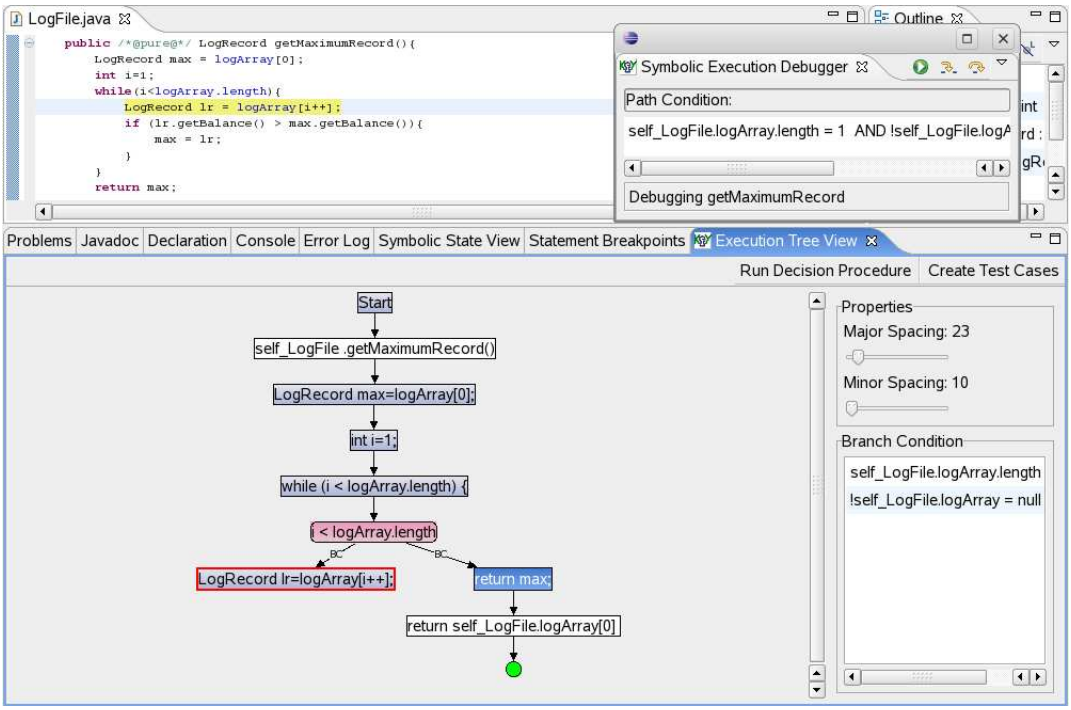

Figure 3.5: *Execution Tree View* of the *Symbolic Execution Debugger*.

is visualized as a green circle which indicates that execution terminates normally, i.e no uncaught exception is thrown. The path condition for the currently selected node is shown in the *Symbolic Execution Debugger View*. The local tool bar of the *Symbolic Execution Debugger View* is equipped with buttons for controlled symbolic execution, namely *Run*, *Step Into*and *Step Over*.

The execution tree in Figure 3.5 can be expanded step-wise, for instance the application of *Step Into* yields the partial execution tree in Figure 3.6.

The command *Step Into* can be applied iteratively in order to discover particular execution paths, like the execution path depicted in Figure 3.7. .

In the execution tree, symbolic states are just represented by their program counters, the *Symbolic State View* is responsible for visualizing entire symbolic states. The visualization of the state represented by the bottom circle in Figure 3.7 is depicted in Figure 3.8.

In general, a symbolic state is visualized by a set of so-called *symbolic object diagrams*. A particular *symbolic object diagram* can be chosen with the sliders **Instance Configuration** and **Index Configuration**. The slider **Instance Configuration** iterates over the possible instances and their associations. On the other hand, the slider **Index Configuration** iterates over the used indices in arrays. *Symbolic object*

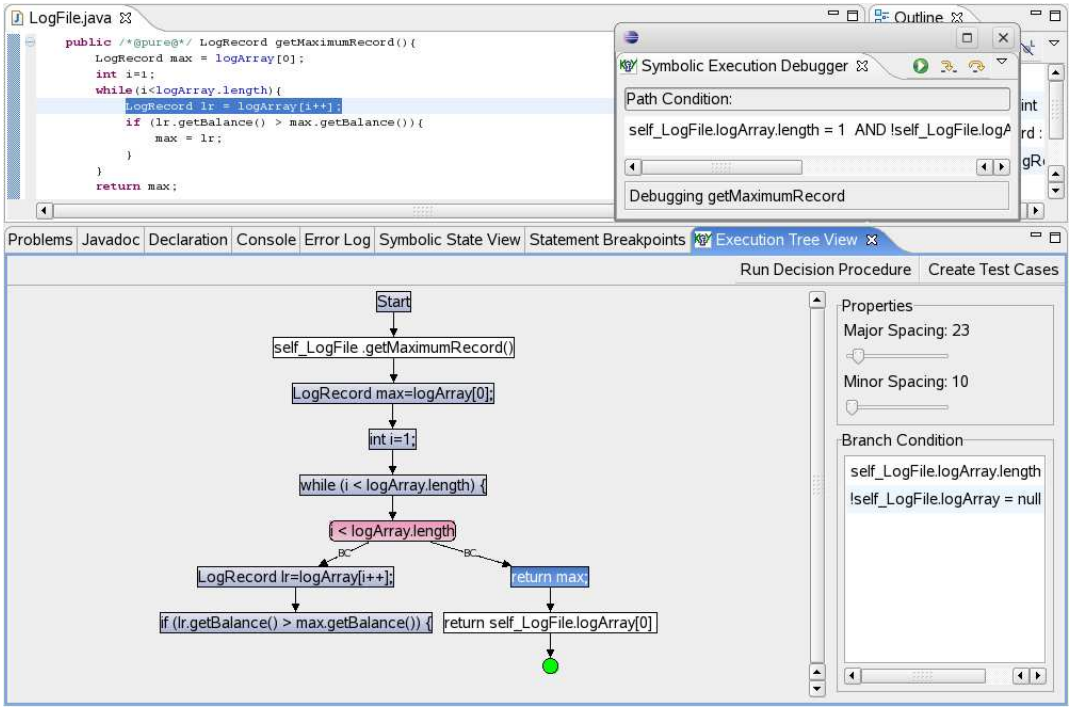

Figure 3.6: *Execution Tree View* after the application of *Step Into* to the execution tree that is depicted in Figure 3.5.

diagrams are created for the prestate and poststate which is associated with the particular execution path (adjustable with the checkboxes **Prestate** and **Poststate**). The method `getMaximumRecord()` does not change the state, so the *symbolic object diagrams* for the prestate and poststate are equal. Primitive attributes in a *symbolic object diagram* are attached with constraints which hold in the prestate. The constraints for the current selected attribute are shown in the constraint compartment on the right of the *Symbolic State View*.

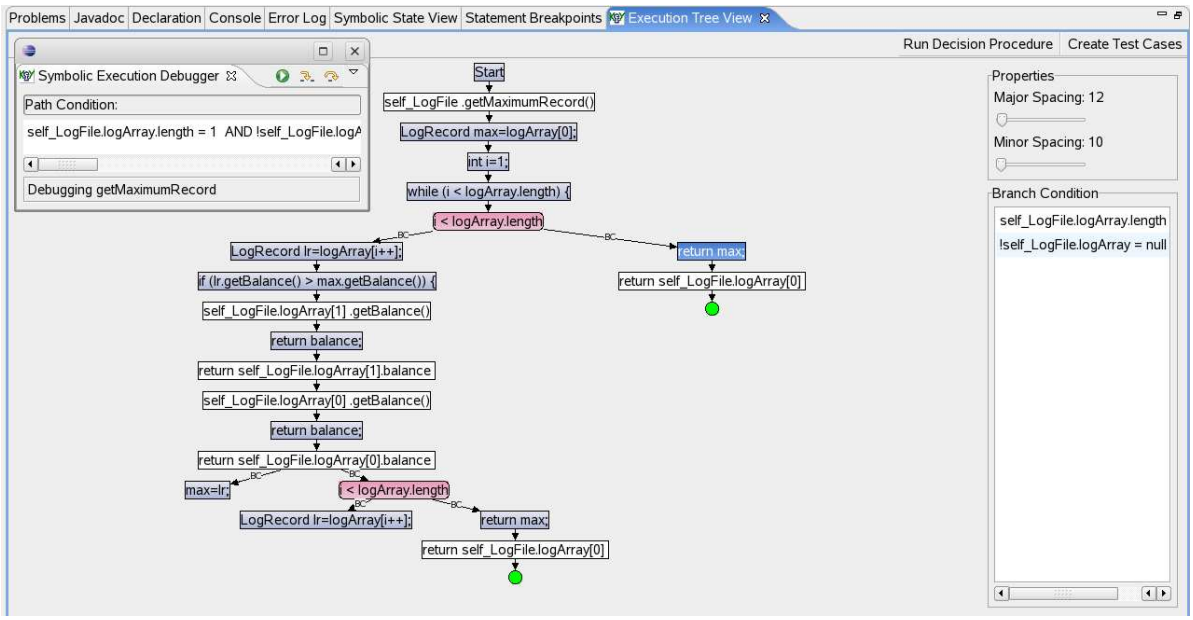

Figure 3.7: *Execution Tree View.*

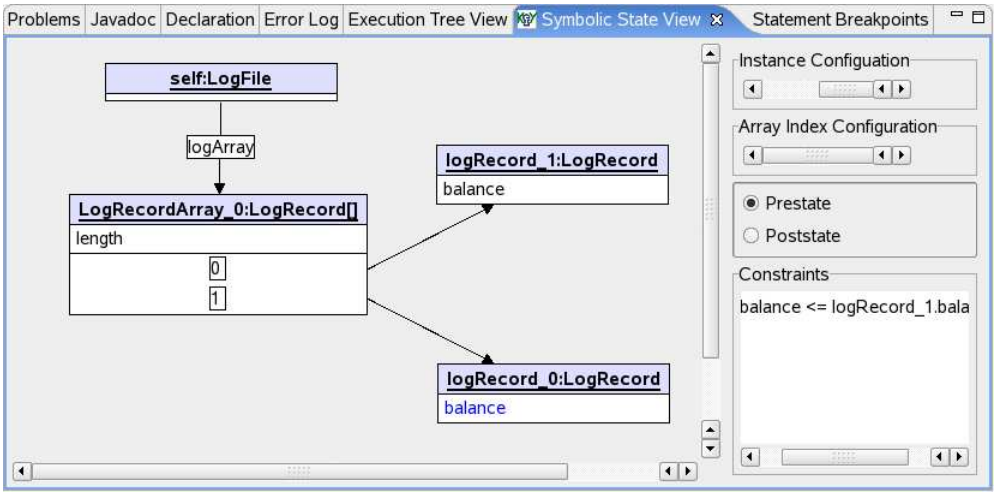

Figure 3.8: *Symbolic object diagram* displayed by the *Symbolic Execution Debugger*.